2000-07

Secure Introduction of One-way Functions

Volpano, Dennis

Proc. 13th IEEE Computer Security Foundations Workshop, pp. 246-254, Cambridge UK, July 2000.
http://hdl.handle.net/10945/35274

Secure Introduction of One-way Functions

Dennis Volpano
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943, USA
Email: volpano@cs.nps.navy.mil

Abstract

Conditions are given under which a one-way function can be used safely in a programming language. The security proof involves showing that secrets cannot be leaked easily by any program meeting the conditions unless breaking the one-way function is easy. The result is applied to a password system where passwords are stored in a public file as images under a one-way function.[1]

1. Introduction

One-way functions play an important role in security. Roughly speaking, a function f is one-way if for all w, it is easy to compute $f(w)$ but hard to find a z, given $f(w)$, such that $f(z) = f(w)$. One-way functions come in different flavors. Some are permutations, while others are hash functions. They operate upon an arbitrary-length pre-image message, producing what is called a *message digest*. A message digest may have fixed length. Examples of hash functions include, MD5, which produces a 128-bit digest, and SHA1, which yields a 160-bit digest [3]. The hardness property coupled with fixed-length digests make certain one-way hash functions appealing for storing passwords on systems and creating pre-images of digital signatures. The main result of this paper is independent of the flavors of one-way functions.

A related property is claw-freeness [2]. A hash function f is said to be *claw-free* if it is hard to find a pair (x, y), where $x \neq y$, such that $f(x) = f(y)$. For a small message space, a hash function may be one-way but fail to be claw-free due to a birthday attack. The basic idea is that one can significantly reduce the size of a message space and still expect to find, with reasonable probability, two messages that collide. Whether this is an issue depends on the application. This paper is not concerned with the claw-free property.

In this paper, we are interested in identifying conditions under which a one-way function can be used in a programming language safely and with more flexibility than what an information-flow property like Noninterference [7] allows. For instance, a cryptographic API for a programming language might include MD5. In this case, the conditions should make leaking a secret using MD5 in any program as hard as inverting MD5. This is a security property under which we justify downgrading MD5 message digests.

We start with the definition of one-way functions from [4]. A function $f : \Sigma^* \longrightarrow \Sigma^*$ is one-way if

1. $|w| = |f(w)|$ for all w (f is length preserving),
2. f is computable in polynomial time, and
3. for every probabilistic polynomial time Turing machine M, every k, and sufficiently large n, if we pick a random w of length n and run M on input $f(w)$,

$$\Pr[M(f(w)) = y \text{ where } f(y) = f(w)] \leq n^{-k}.$$

The first and second conditions are irrelevant as far as our main result is concerned. The probability in the third condition is taken over the random choices made by M and the random choice of w. The third condition effectively merges two properties that we need to distinguish for the purpose of constructing a security proof. One is simply the likelihood that f avoids collisions with respect to a given input distribution. This property we term *collision resistance*. The other is purely a property about inversion where the third condition becomes $\Pr[M(f(w)) = w] \leq n^{-k}$. This is the *one-wayness* property of f.

If string w is considered private (high) then we might argue that $f(w)$ could be considered public (low) based on the one-wayness of f. However, it is actually unsound to do so unless care is taken in what we allow as arguments to f. For instance, suppose f is a one-way function, variable h stores a k-bit password, and *mask* is a low variable. Then consider

[1] To appear at the 13th IEEE Computer Security Foundations Workshop, Cambridge, England, 3-5 July, 2000.

```
l := 0;
mask := 2^{k-1};
while mask ≠ 0 do
    if f(h) = f(h | mask) then
        l := l | mask;
    mask := mask ≫ 1
```

Figure 1. An efficient leak of h

the code in Figure 1. It copies (leaks) h to low variable l in time linear in k. (It might fail to copy every bit of h because of collisions but this may be unlikely depending on the collision resistance of f.)

However, there are practical examples of where we need to treat a message digest as low. Consider a challenge-response protocol. A participant may respond publicly with a message digest computed over a shared secret and a public challenge it receives. We want the digest to be treated as low. Another example is password checking. If h stores a password then a simple password checker is given by the assignment

$$b := (f(h) = f(r))$$

where b is a low output variable and r is the input to the checker. We would expect r and h to be high variables. After all, r may match h, and indeed usually will. However, the result of comparing the message digests must be low.

So we want a set of conditions for a programming language that prohibits abuses of one-way functions, as in Figure 1, yet recognizes legitimate downgrading by them in other situations. This paper describes such a set of conditions via a type system. Further, we need a sense in which these conditions are sound. They are certainly not sound with respect to Noninterference [7] due to downgrading. However, they are sound in the following sense. It can be proved that leaking the secret contents of a variable h using any program P that meets the conditions is as hard as learning h with a program where access to h is prohibited, but the program can access $f(h)$, call f on inputs of its choice and flip a coin. And deducing h in this context clearly amounts to inverting $f(h)$ using a probabilistic Turing machine. By the one-wayness of f then, we expect P to succeed with very low probability in polynomial time, for sufficiently-long and uniformly-distributed values of h.

Informally, we reduce the problem of inverting a one-way function to that of leaking a secret h via a well-typed program. We begin with a well-typed program that can access h directly and show that its low computation can be simulated by a program with no references to high variables except in calls of the form $f(h)$ and in comparisons of the form $f(h) = f(r)$, for a high read-only variable r. The latter comparisons are then eliminated by an independent random variable whose distribution is governed by the collision resistance of f with respect to the well-typed program's input distribution. (It is irrelevant that we may not know the distribution because the reduction only relies upon its existence.) The result is a program that uses f, $f(h)$, and an independent random variable to simulate the well-typed program's low computation with at least the same probability of success and with at most a constant increase in time complexity. Therefore, any bound on the probability of finding h from $f(h)$ within polynomial time can apply to the probability of leaking h with a well-typed polynomial-time command. This is a security property that applies, for instance, to the simple password checker above.

2. The language and semantics

A program is expressed in an imperative language:

$$\begin{array}{llll} (\textit{expr}) & e & ::= & x \mid h \mid n \mid f(e) \mid f(h) = f(r) \mid \\ & & & e_1 + e_2 \mid e_1 < e_2 \mid e_1 = e_2 \mid \\ & & & e_1 \,\&\, e_2 \mid e_1 \gg e_2 \mid (e_1 \mid e_2) \\ (\textit{cmds}) & c & ::= & \textbf{skip} \mid x := e \mid c_1; c_2 \mid \\ & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } e \textbf{ do } c \end{array}$$

Metavariable x ranges over identifiers that are mapped by memories to integers, n ranges over integer literals, f is a function mapping integers to integers, and h and r are read-only variables. There are three bitwise operators ($\&, \gg, |$). Integers are the only values; we use 0 for false and nonzero for true.

A standard transition semantics for the language is given in Figure 2. It is completely deterministic and defines a transition function $\longrightarrow$ on configurations. A *memory* μ is a mapping from variables to integers. A configuration is either a pair (c, μ) or simply a memory μ. In the first case, c is the command yet to be executed; in the second case, the command has terminated, yielding final memory μ. As usual, we define $\kappa \longrightarrow^0 \kappa$, for any configuration κ, and $\kappa \longrightarrow^k \kappa''$, for $k > 0$, if there is a configuration κ' such that $\kappa \longrightarrow^{k-1} \kappa'$ and $\kappa' \longrightarrow \kappa''$.

Expressions are evaluated atomically and we extend the application of μ to expressions, writing $\mu(e)$ to denote the value of expression e in memory μ. We say that $\mu(f(e)) = f(\mu(e))$, $\mu(e_1 + e_2) = \mu(e_1) + \mu(e_2)$, and so on. The other expressions are handled similarly. Note that $\mu(e)$ is defined for all e, as long as every identifier in e is in $\mathit{dom}(\mu)$.

2.1. Probabilistic execution

In our reduction of Section 4, we talk about the probabilistic simulation of a command with respect to a joint distribution d for its free variables (d is finite for a given command if memories are mappings to k-bit integers for a

(NO-OP) $$(\textbf{skip}, \mu) \longrightarrow \mu$$

(UPDATE) $$\frac{x \in \mathit{dom}(\mu)}{(x := e, \mu) \longrightarrow \mu[x := \mu(e)]}$$

(SEQUENCE) $$\frac{(c_1, \mu) \longrightarrow \mu'}{(c_1; c_2, \mu) \longrightarrow (c_2, \mu')}$$

$$\frac{(c_1, \mu) \longrightarrow (c_1', \mu')}{(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')}$$

(BRANCH) $$\frac{\mu(e) \neq 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_2, \mu)}$$

(LOOP) $$\frac{\mu(e) = 0}{(\textbf{while } e \textbf{ do } c, \mu) \longrightarrow \mu}$$

$$\frac{\mu(e) \neq 0}{(\textbf{while } e \textbf{ do } c, \mu) \longrightarrow (c; \textbf{while } e \textbf{ do } c, \mu)}$$

Figure 2. Transition semantics

fixed k). A simulation may need to flip a coin but this occurs only once, at the start of an execution, and therefore we can achieve the effect by introducing an independent random variable as input to the simulation. Although this keeps the simulation deterministic, it still calls for attention in the semantics because the input variable must be initialized from a probability space prior to execution, apart from other free variables [1].

The free and random variables of a command can be treated uniformly as just free variables if a command is represented as a discrete Markov chain, the states of which are configurations [5]. The idea is to execute a command simultaneously in all memories that map exactly its free variables. For each such memory μ, it begins execution in μ with probability $d(\mu)$. An execution then becomes a sequence of probability measures on configurations. The stochastic matrix T of the Markov chain in this case is trivial; each row of the matrix is a point mass. That means there is no splitting of mass after execution begins, only accumulation of it (cf. pg. 337 of [1]). Each measure in a sequence is determined by taking the linear transformation of the immediately preceding measure with respect to T. See [5] for details.

For example, execution of $y := \neg x$ is given in Figure 3 relative to a particular joint distribution for the four possible memories. We say that $y := \neg x$ terminates in memory $[x := 0, y := 1]$ in one step with unconditional probability $5/8$ and in $[x := 1, y := 0]$ in one step with probability $3/8$.

As another example, consider the loop

$$\textbf{while } x \textbf{ do } x := \neg x$$

whose execution is given in Figure 4 for a particular distribution. Mass accumulates at $[x := 0]$ (or $(\{\}, [x := 0])$ in the notation of [5]) in the final step. We say the command terminates in $[x := 0]$ in three steps with unconditional probability 1.

In general, suppose μ is a memory, c has free variables $x_1, \ldots, x_n$ and $\mu_1, \ldots, \mu_m$ are memories with domain $x_1, \ldots, x_n$ from which c terminates in μ in at most k steps. If d is a joint distribution for $x_1, \ldots, x_n$, then we say c terminates in μ in k steps with unconditional probability $d(\mu_1) + \cdots + d(\mu_m)$.

3. The type system

Following previous work, the types are as follows:

$$\begin{array}{lll} \textit{(data types)} & \tau & ::= L \mid H \\ \textit{(phrase types)} & \rho & ::= \tau \mid \tau \textit{ var} \mid \tau \textit{ cmd} \end{array}$$

The data types are just the security levels low and high. The rules of the type system are given in Figure 5. Here γ is a typing that maps variables (perhaps read only) to types of the form τ *var* or τ. If $\gamma(x) = \tau$ then we say that x is a *read-only variable* in γ. We distinguish h and r as special read-only variables in that $\gamma(h) = H = \gamma(r)$, for all γ.

The typing rules for the other binary operators are similar to that for EQ. Notice that where downgrading is taking place, specifically in rules QUERY and IMAGE, it is done with respect to read-only variables, namely r and h. This is key to getting a reduction. It is these two rules that break traditional Noninterference. Rule IMAGE comes in handy when typing the code of a challenge-response protocol, in particular, the C code that makes up the GNU implementation of CHAP. It allows a low digest to be computed over a challenge and a secret, the concatenation of which is the value of h. Rule QUERY is useful in password-checking contexts. More is said about these applications in Section 5.

Notice that the code in Figure 1 is not well typed. Expression $f(h \mid \textit{mask})$ can only be typed using rule HASH, forcing it to have type H since $\gamma(h) = H$ for all γ. But then the guard of the conditional has type H while its body has type L *cmd* which cannot be reconciled.

4. The reduction

The basic idea is to show that every well-typed command's low computation can be simulated, with at most a constant increase in time complexity, by a command whose only references to high variables are in calls to f. However, we are not finished. The simulation still has calls of the form $f(h)$ and $f(r)$. Instances of $f(h)$ can remain because they form the input to a command (the adversary) for computing h, but all calls $f(r)$ must be eliminated.

We begin with some definitions:

Definition 4.1 *Memories μ and ν are equivalent with respect to a typing γ, written $\mu \sim_\gamma \nu$, if $\mu(h) = \nu(h)$ and $\mu(x) = \nu(x)$ for all x where $\gamma(x) = L$ var or $\gamma(x) = L$.*

Definition 4.2 *We say that c is a low command with respect to γ if the only occurrences of high variables in c with respect to γ are references to h in $f(h)$.*

Definition 4.3 *Given a joint distribution d on dom(γ), we say that command c' is a low probabilistic simulation of a command c, relative to γ and d, if c' is a low command with respect to γ, and if c terminates in μ in k steps with unconditional probability q, relative to d, then there is a memory ν such that c' terminates in ν in at most $k+1$ steps with probability q', $q' \geq q$ and $\nu \sim_\gamma \mu$.*

We will need the following lemma:

Lemma 4.1 *Suppose c is a well-typed command with respect to γ and that it has no occurrence of $f(h) = f(r)$. Then there is a low command c' with respect to γ such that*

$$
\begin{gathered}
(y := \neg x, [x := 0, y := 0]) : \tfrac{3}{8} \\
(y := \neg x, [x := 0, y := 1]) : \tfrac{1}{4} \\
(y := \neg x, [x := 1, y := 0]) : \tfrac{1}{8} \\
(y := \neg x, [x := 1, y := 1]) : \tfrac{1}{4} \\
\downarrow \\
[x := 0, y := 1] : \tfrac{3}{8} + \tfrac{1}{4} = \tfrac{5}{8} \\
[x := 1, y := 0] : \tfrac{1}{8} + \tfrac{1}{4} = \tfrac{3}{8}
\end{gathered}
$$

Figure 3. Execution of $y := \neg x$ as a sequence of measures

$$
\begin{gathered}
(\textbf{while } x \textbf{ do } x := \neg x, [x := 0]) : \tfrac{1}{3} \\
(\textbf{while } x \textbf{ do } x := \neg x, [x := 1]) : \tfrac{2}{3} \\
\downarrow \\
[x := 0] : \tfrac{1}{3} \\
(x := \neg x;\ \textbf{while } x \textbf{ do } x := \neg x, [x := 1]) : \tfrac{2}{3} \\
\downarrow \\
[x := 0] : \tfrac{1}{3} \\
(\textbf{while } x \textbf{ do } x := \neg x, [x := 0]) : \tfrac{2}{3} \\
\downarrow \\
[x := 0] : \tfrac{1}{3} + \tfrac{2}{3} = 1
\end{gathered}
$$

Figure 4. Execution of $\textbf{while } x \textbf{ do } x := \neg x$ as a sequence of measures

(INT) $\gamma \vdash n : L$

(IMAGE) $\gamma \vdash f(h) : L$

(QUERY) $\gamma \vdash f(h) = f(r) : L$

(CONST) $$\frac{\gamma(x) = \tau}{\gamma \vdash x : \tau}$$

(R-VAL) $$\frac{\gamma(x) = \tau\ \mathit{var}}{\gamma \vdash x : \tau}$$

(EQ) $$\frac{\gamma \vdash e_1 : \tau,\ \ \gamma \vdash e_2 : \tau}{\gamma \vdash e_1 = e_2 : \tau}$$

(HASH) $$\frac{\gamma \vdash e : \tau}{\gamma \vdash f(e) : \tau}$$

(SKIP) $\gamma \vdash \textbf{skip} : H\ \mathit{cmd}$

(ASSIGN) $$\frac{\gamma(x) = \tau\ \mathit{var},\ \ \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ \mathit{cmd}}$$

(COMPOSE) $$\frac{\gamma \vdash c_1 : \tau\ \mathit{cmd},\ \ \gamma \vdash c_2 : \tau\ \mathit{cmd}}{\gamma \vdash c_1; c_2 : \tau\ \mathit{cmd}}$$

(IF) $$\frac{\gamma \vdash e : \tau,\ \ \gamma \vdash c_1 : \tau\ \mathit{cmd},\ \ \gamma \vdash c_2 : \tau\ \mathit{cmd}}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ \mathit{cmd}}$$

(WHILE) $$\frac{\gamma \vdash e : \tau,\ \ \gamma \vdash c : \tau\ \mathit{cmd}}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau\ \mathit{cmd}}$$

(BASE) $L \subseteq H$

(REFLEX) $\rho \subseteq \rho$

(CMD$^-$) $$\frac{\tau_1 \subseteq \tau_2}{\tau_2\ \mathit{cmd} \subseteq \tau_1\ \mathit{cmd}}$$

(SUBTYPE) $$\frac{\gamma \vdash p : \rho_1,\ \ \rho_1 \subseteq \rho_2}{\gamma \vdash p : \rho_2}$$

Figure 5. Typing rules

for all μ where dom(μ) = dom(γ), whenever $(c, \mu) \longrightarrow^n \mu'$, there is a μ'' and m such that $(c', \mu) \longrightarrow^m \mu''$, $\mu' \sim_\gamma \mu''$ and $m \leq n$.

A proof of this lemma can be obtained by modifying the proof of Theorem 5.1 in [6] in order to treat the slightly-different notion of memory equivalence used here and to handle f calls in expressions.

Finally, the reduction is given by the following theorem:

Theorem 4.2 *If c is a well-typed command with respect to γ and d is a joint distribution on dom(γ), then c has a low probabilistic simulation relative to γ and d.*

Proof. There are two cases, one where c has no instances of $f(h) = f(r)$ and the other where it does. First suppose that c has no occurrence of $f(h) = f(r)$. Then let c' be the low command given by Lemma 4.1 for c. We can show that c' is a low probabilistic simulation of c as follows.

Let d be a joint distribution on *dom*(γ) and let

$$M = \{\mu \mid \mathit{dom}(\mu) = \mathit{dom}(\gamma)\}.$$

Suppose c terminates in a memory μ in k steps with unconditional probability q relative to d. Let $\mu_1, \ldots, \mu_n$ be all memories in M for which $(c, \mu_i) \longrightarrow^j \mu$ for some j where $j \leq k$. Then

$$q = d(\mu_1) + d(\mu_2) + \cdots + d(\mu_n).$$

By Lemma 4.1, there is a μ_i' and m_i for each μ_i such that $(c', \mu_i) \longrightarrow^{m_i} \mu_i'$, $\mu_i' \sim_\gamma \mu$ and $m_i \leq j$. Let ν_i be μ_i such that *dom*(ν_i) contains exactly h and all low variables of γ. Since c' is low, there is a ν_i' such that $(c', \nu_i) \longrightarrow^{m_i} \nu_i'$, $\nu_i' \sim_\gamma \mu_i'$, and *dom*($\nu_i'$) = *dom*($\nu_i$), for $i = 1, \ldots, n$. By transitivity of $\sim_\gamma$,

$$\mu_1' \sim_\gamma \mu_2' \sim_\gamma \cdots \sim_\gamma \mu_n'.$$

Therefore, $\nu_1' = \nu_2' = \cdots = \nu_n'$. So let $\nu = \nu_1'$. And c' terminates in ν in at most *max*($m_1, \ldots, m_n$) steps with unconditional probability at least q if for any memory ν', whose domain contains exactly h and all low variables of γ, c' begins execution in ν' with probability

$$\sum_{\{\mu \in M \mid \mu \sim_\gamma \nu'\}} d(\mu).$$

Also, *max*($m_1, \ldots, m_n$) $\leq k$.

Now suppose c has an occurrence of $f(h) = f(r)$. Without loss of generality, assume c has the form

$$\textbf{if } f(h) = f(r) \textbf{ then } c_1 \textbf{ else } c_2$$

where c_1 and c_2 have no instances of $f(h) = f(r)$. There is no loss of generality here because $f(h) = f(r)$ is constant in any memory, given that h and r are read-only variables in every typing, and c is well typed under γ. So let c_1' and c_2' be the low commands given by Lemma 4.1 for c_1 and c_2 respectively. We can show that the command c' given by

$$(\textbf{if } X \textbf{ then } c_1' \textbf{ else } c_2');\ X := 0$$

where X is an independent boolean random variable not in *dom*(γ), is a low probabilistic simulation of c.

Suppose, relative to d, that c_1 terminates in μ in fewer than k steps with probability q_1 given that $f(h) = f(r)$. Since there is no free occurrence of r in c_1, it also terminates in μ in fewer than k steps with probability q_1 given that $f(h) \neq f(r)$. Therefore, q_1 is an unconditional probability that c_1 terminates in μ in fewer than k steps. Likewise, suppose c_2 terminates in μ in fewer than k steps with probability q_2 given that $f(h) \neq f(r)$. Since there is no free occurrence of r in c_2, it also terminates in μ in fewer than k steps with probability q_2 given that $f(h) = f(r)$. So q_2 is an unconditional probability that c_2 terminates in μ in fewer than k steps. Therefore, c terminates in μ in k steps with probability

$$q = p \cdot q_1 + (1 - p) \cdot q_2$$

where p is defined by

$$\sum_{\{\mu \in M \mid f(\mu(h)) = f(\mu(r))\}} d(\mu).$$

From above, there are memories ν_1' and ν_2', each equivalent to μ, such that c_1' terminates in ν_1' in fewer than k steps with probability q_1', c_2' terminates in ν_2' in fewer than k steps with probability q_2', $q_1' \geq q_1$ and $q_2' \geq q_2$. By the transitivity of $\sim_\gamma$, $\nu_1' \sim_\gamma \nu_2'$ which implies $\nu_1' = \nu_2'$ since neither has in its domain a high variable of γ besides h.

Now if $(c_1', \nu_1) \longrightarrow^j \nu_1'$, for some j and ν_1, then $(c_1', \nu_1[X := n]) \longrightarrow^j \nu_1'[X := n]$ because X does not occur free in c_1'. Likewise for c_2'. And if c' terminates, it does so in a memory that maps X to 0. Therefore, take $\nu = \nu_1'[X := 0]$, and we have that $\nu_1'[X := 0] \sim_\gamma \mu$ since $\nu_1'[X := 0] \sim_\gamma \nu_1'$ and $\nu_1' \sim_\gamma \mu$.

Finally, the unconditional probability that c' terminates in ν in at most $k + 1$ steps is the probability that

$$\textbf{if } X \textbf{ then } c_1' \textbf{ else } c_2'$$

terminates in ν_1' in at most k steps. And because X is independent of *dom*(γ), this latter probability is given by

$$q' = p \cdot q_1' + (1 - p) \cdot q_2'$$

if for any memory ν', whose domain contains exactly h and all low variables of γ, c' begins execution in $\nu'[X := 1]$ with probability

$$p \cdot \left(\sum_{\{\mu \in M \mid \mu \sim_\gamma \nu'\}} d(\mu) \right).$$

Finally, $q' \geq q$. $\square$

Suppose γ is a typing with a low variable l in its domain, d is a distribution on $dom(\gamma)$ and c is a command for copying h to l that is well-typed relative to γ. Now suppose we run c simultaneously in all memories whose domains are equal to $dom(\gamma)$ for $p(n)$ steps according to the input distribution d, where p is a polynomial and n is the length of the binary encoding of a memory. And suppose that after $p(n)$ steps, c terminates in a memory μ where $\mu(l) = \mu(h)$ with probability q. By Theorem 4.2, there is a low command c' that terminates in no more than $p(n) + 1$ steps in a memory ν where $\nu \sim_\gamma \mu$ with probability at least q. Furthermore, $\nu(l) = \nu(h)$ since $\nu \sim_\gamma \mu$. And because c' is low, it has therefore managed to find h without any high variables as input, just occurrences of $f(h)$ is all. This brings us to the following Corollary:

Corollary 4.3 *Any bound on the probability of finding h from $f(h)$ within polynomial time, for a particular integer size and distribution on h, also applies to the probability of leaking h with a well-typed command in polynomial time with respect to that size and distribution.*

Notice that probability p in the preceding proof takes into account the probability that $h = r$ as well as the collision resistance of f. Indeed, we would expect our simple password checker to be run with fairly high probability in a memory where $h = r$ if h stores a password and r is the checker's input. The reduction says that any well-typed program that attempts to exploit this fact has no advantage over a program that cannot reference h or r, but instead can access $f(h)$, call f on inputs of its choice and flip a coin. The one-wayness of f is treated by allowing instances of $f(h)$ in a low probabilistic simulation, which is a program squarely within the realm of a probabilistic model of computation used to define a one-way function [4].

5. Application to password systems

Consider again our simple password checker

$$b := (f(h) = f(r))$$

where variable h stores a password, b is an output variable and r is the input to the checker. Now we want to argue that the checker is secure. We begin by asserting what we know about the free variables. Well, since the output of the checker is public, we expect b to be low. On the other hand, h stores a password so it should be high. Under normal use of the checker, r will likely store the contents of h, and since h is high, we assert that r is high as well. Furthermore, the checker doesn't attempt to update h or r and therefore is well typed under the assumption that these variables are read only. So the checker is secure in the sense that it belongs to a class of programs for which the complexity of leaking h rests upon the intractability of inverting $f(h)$ for sufficiently-long and uniformly-distributed values of h, by the above Corollary. The checker's low probabilistic simulation is given by

$$(\textbf{if } X \textbf{ then } b := 1 \textbf{ else } b := 0);\ X := 0$$

where X is the random variable in the proof of Theorem 4.2.

Now suppose passwords are stored in a read-protected file in the clear as in, for example, a secrets file for CHAP (Cryptographic Handshake Authentication Protocol) widely used by PPP. In this case, the checker becomes just

$$b := (h = r)$$

We can argue that this checker too is secure using the reduction in Theorem 4.2 where we assume f is the identity function. But this assumption requires that rule IMAGE be eliminated, for clearly it is no longer sound. This means the adversary can no longer access the "resource" $f(h)$. Instead, we replace this form of access to h with a new form, namely *match*(h, e), which is true in μ if $\mu(h) = \mu(e)$. It has the following typing rule:

$$\frac{\gamma \vdash e : L}{\gamma \vdash \textit{match}(h, e) : L}$$

Again, there is downgrading taking place, as in rule IMAGE. Whether *match* has any utility from the standpoint of writing useful programs is not important. What is important is that we provide the adversary with the resources we would realistically expect it to have. In the case of one-way functions, the adversary expects $f(h)$, but with f treated as the identity function, it now becomes the ability to match inputs of the adversary's choice against h which is precisely what *match* provides.

If access to h is limited to *match* queries and the values of h are uniformly distributed k-bit integers, then the probability of successfully leaking h with any deterministic polynomial-time command containing an independent random variable goes to zero as k increases [6]. If rule QUERY is replaced by the rule

$$\gamma \vdash h = r : L$$

then the second checker is well typed in the modified system, and is therefore secure in the sense that it belongs to a class of programs for which the complexity of leaking h rests upon this asymptotic hardness result, by Theorem 4.2.

Finally, to say something about the password system as a whole, we need to treat password updates as well. A password updater for h is given in Figure 6. The updater expects the old password, so free variable *old* is asserted to be a high

```
if f(h) = f(old) then
    check strength of new password
    h := new;
else skip
```

Figure 6. A password update program for h

variable, as is *new* which stores the new password. This program is well typed in a different type system, namely that of [7], assuming the strength-checking portion is well typed. Therefore, it satisfies a Noninterference property which is appropriate for this program, as there is no downgrading taking place.

The results here can also be applied to the GNU implementation of CHAP. The C code that hashes randomly-generated server challenges with a shared CHAP secret, using RSA's MD5, is well typed. That tells us the code belongs to a class of programs for which leaking shared secrets is as hard as inverting 16-byte MD5 message digests computed over random challenges and sufficiently-long and uniformly-distributed CHAP secrets. It is really only in this sense that one can argue the code "protects" the confidentiality of shared CHAP secrets.

One final word is needed about modeling adversaries. We can identify two kinds of adversaries: inside and outside. Inside adversaries write programs that we want to trust and have direct access to secrets like h and r. Outside adversaries write programs we never trust, and therefore are denied direct access to secrets through some sort of access control. Each adversary has a typing rule where downgrading occurs. For the outside adversary, it is rule IMAGE (or the rule for *match* if f is the identity) and for the inside adversary, it is rule QUERY. Both forms of adversary should be represented in a computational model. One could argue that the work in [6] does not treat inside adversaries completely because it does not consider a rule like QUERY.

6. Conclusion

This paper presents syntactic conditions, via a type system, for introducing one-way functions into a programming language with more flexibility than what Noninterference allows. These conditions are sound in a computational sense and allow one to argue for the security of some systems where downgrading must occur.

Notice that functions are not part of the language we considered. That means commands in the language cannot call other commands. Functions pose a problem since λ-bound variables, although constant in a function body, can be bound in different ways through different function applications. This capability breaks the reduction. A useful line of work would be to identify conditions under which functions could be introduced securely.

7. Acknowledgments

I would like to thank Geoffrey Smith for his comments on the paper. This material is based upon activities supported by the National Science Foundation under Agreement No. CCR-9900909.

References

[1] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.

[2] R. Rivest. *Cryptography*, volume A of *Handbook of Theoretical Computer Science*, chapter 13. The MIT Press/Elsevier, 1990.

[3] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1996. Second Edition.

[4] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[5] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, 1999.

[6] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proceedings 27th Symposium on Principles of Programming Languages*, pages 268–276, Boston, MA, Jan. 2000.

[7] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.